EXHIBIT C

```
/*  *******************************************************************
    File: sslrec.c

    SSL Plus: Security Integration Suite(tm)
    Version 1.1.1 -- August 11, 1997

    Copyright (c)1996, 1997 by Consensus Development Corporation
          Copyright (c)1997, 1998 by Aventail Corporation

    Portions of this software are based on SSLRef(tm) 3.0, which is
    Copyright (c)1996 by Netscape Communications Corporation. SSLRef(tm)
    was developed by Netscape Communications Corporation and Consensus
    Development Corporation.

    In order to obtain this software, your company must have signed
    either a PRODUCT EVALUATION LICENSE (a copy of which is included in
    the file "LICENSE.TXT"), or a PRODUCT DEVELOPMENT LICENSE. These
    licenses have different limitations regarding how you are allowed to
    use the software. Before retrieving (or using) this software, you
    *must* ascertain which of these licenses your company currently
    holds. Then, by retrieving (or using) this software you agree to
    abide by the particular terms of that license. If you do not agree
    to abide by the particular terms of that license, than you must
    immediately delete this software. If your company does not have a
    signed license of either kind, then you must either contact
    Consensus Development and execute a valid license before retrieving
    (or using) this software, or immediately delete this software.

    *******************************************************************

    File: sslrec.c     Encryption, decryption and MACing of data

    All the transformations which occur between plaintext and the
    secured, authenticated data that goes out over the wire. Also,
    detects incoming SSL 2 hello messages and hands them off to the SSL 2
    record layer (and hands all SSL 2 reading & writing off to the SSL 2
    layer).

    ***************************************************************** */

/* #define HYPER_DEBUG 1 */

#ifdef HYPER_DEBUG
#include <stdio.h>
#endif

#ifndef _SSL_H_
#include "ssl.h"
#endif

#ifndef _SSLREC_H_
#include "sslrec.h"
#endif

#ifndef _SSLALLOC_H_
#include "sslalloc.h"
#endif

#ifndef _CRYPTYPE_H_
#include "cryptype.h"
#endif

#ifndef _SSLCTX_H_
#include "sslctx.h"
#endif

#ifndef _SSLALERT_H_
#include "sslalert.h"
#endif

#ifndef _SSL2_H_
```

```
#include "ssl2.h"
#endif

#include <string.h>

static SSLErr DecryptSSLRecord(uint8 type, SSLBuffer *payload, SSLContext *ctx);
static SSLErr VerifyMAC(uint8 type, SSLBuffer data, uint8 *compareMAC, uint64 seqNo, SSLContext
    *ctx);
static SSLErr ComputeMAC(uint8 type, SSLBuffer data, SSLBuffer mac, uint64 seqNo, SSLBuffer
    secret, CipherContext *cipherCtx, SSLContext *ctx);
static uint8* SSLEncodeUInt64(uint8 *p, uint64 value);

/* ReadSSLRecord
 *  Attempt to read & decrypt an SSL record.
 */
SSLErr
SSLReadRecord(SSLRecord *rec, SSLContext *ctx)
{   SSLErr      err;
    uint32      len, contentLen;
    uint8           *progress;
    SSLBuffer       readData, cipherFragment;

#ifdef HYPER_DEBUG
   fprintf(stderr, "Got into SSLReadRecord, whee!\n");
#endif

   /* if we get UDP data when we aren't expecting it, that's really bad,
      so report an appropriate error. */
   if((rec->contentType == SSL_application_data_ssloppy) &&
      (! ctx->ssloppy))
          return SSLProtocolErr;



   if (!ctx->partialReadBuffer.data || ctx->partialReadBuffer.length < 5)
     {   if (ctx->partialReadBuffer.data)
             if ((err = SSLFreeBuffer(&ctx->partialReadBuffer, &ctx->sysCtx)) != 0)
             {   SSLFatalSessionAlert(alert_close_notify, ctx);
                 return ERR(err);
             }
         if ((err = SSLAllocBuffer(&ctx->partialReadBuffer, DEFAULT_BUFFER_SIZE, &ctx->sysCtx))
   != 0)
         {   SSLFatalSessionAlert(alert_close_notify, ctx);
             return ERR(err);
         }
     }

    if (ctx->protocolVersion == SSL_Version_Undetermined ||
        ctx->protocolVersion == SSL_Version_3_0_With_2_0_Hello)
        if (ctx->amountRead < 1)
        {   readData.length = 1 - ctx->amountRead;
            readData.data = ctx->partialReadBuffer.data + ctx->amountRead;
            len = readData.length;
            if (ERR(err = ctx->ioCtx.read(readData, &len, ctx->ioCtx.ioRef)) != 0)
            {   if (err == SSLWouldBlockErr)
                    ctx->amountRead += len;
                else
                    SSLFatalSessionAlert(alert_close_notify, ctx);
                return err;
            }
            ctx->amountRead += len;
        }

/* In undetermined cases, if the first byte isn't in the range of SSL 3.0
 *  record types, this is an SSL 2.0 record
 */
    switch (ctx->protocolVersion)
    {   case SSL_Version_Undetermined:
        case SSL_Version_3_0_With_2_0_Hello:
            if (ctx->partialReadBuffer.data[0] < SSL_smallest_3_0_type ||
                ctx->partialReadBuffer.data[0] > SSL_largest_3_0_type)
```

```
                    return SSL2ReadRecord(rec, ctx);
                else
                    break;
            case SSL_Version_2_0:
                return SSL2ReadRecord(rec, ctx);
            default:
                break;
        }


#ifdef HYPER_DEBUG
    fprintf(stderr, "About to get into the read callback stuff\n");
#endif
    if (ctx->amountRead < 5)
    {   readData.length = 5 - ctx->amountRead;
        readData.data = ctx->partialReadBuffer.data + ctx->amountRead;
        len = readData.length;
        if (ERR(err = ctx->ioCtx.read(readData, &len, ctx->ioCtx.ioRef)) != 0)
        {   if (err == SSLWouldBlockErr)
                ctx->amountRead += len;
                         else if (err == SSLIOClosedOverrideGoodbyeKiss && ctx->amountRead ==
  0)
                         {   SSLClose(ctx);
                         return SSLConnectionClosedGraceful;
                         }
                         else
                                         SSLFatalSessionAlert(alert_close_notify, ctx);
                         return err;
                         }
        ctx->amountRead += len;
    }

    ASSERT(ctx->amountRead >= 5);

    progress = ctx->partialReadBuffer.data;
    rec->contentType = *progress++;
    if (rec->contentType < SSL_smallest_3_0_type ||
        rec->contentType > SSL_largest_3_0_type)
        return ERR(SSLProtocolErr);

    rec->protocolVersion = (SSLProtocolVersion)SSLDecodeInt(progress, 2);
    progress += 2;
    contentLen = SSLDecodeInt(progress, 2);
    progress += 2;
    if (contentLen > (16384 + 2048))    /* Maximum legal length of an SSLCipherText payload */
    {   SSLFatalSessionAlert(alert_unexpected_message, ctx);
        return ERR(SSLProtocolErr);
    }

    if (ctx->partialReadBuffer.length < 5 + contentLen)
    {   if ((err = SSLReallocBuffer(&ctx->partialReadBuffer, 5 + contentLen, &ctx->sysCtx)) !=
  0)
        {   SSLFatalSessionAlert(alert_close_notify, ctx);
            return ERR(err);
        }
    }

    if (ctx->amountRead < 5 + contentLen)
    {   readData.length = 5 + contentLen - ctx->amountRead;
        readData.data = ctx->partialReadBuffer.data + ctx->amountRead;
        len = readData.length;
        if (ERR(err = ctx->ioCtx.read(readData, &len, ctx->ioCtx.ioRef)) != 0)
        {   if (err == SSLWouldBlockErr)
                ctx->amountRead += len;
            else
                SSLFatalSessionAlert(alert_close_notify, ctx);
            return err;
        }
        ctx->amountRead += len;
    }

    ASSERT(ctx->amountRead >= 5 + contentLen);

    cipherFragment.data = ctx->partialReadBuffer.data + 5;
    cipherFragment.length = contentLen;

/* Decrypt the payload & check the MAC, modifying the length of the buffer to indicate the
 *  amount of plaintext data after adjusting for the block size and removing the MAC
 *  (this function generates its own alerts)
 */
    if ((err = DecryptSSLRecord(rec->contentType, &cipherFragment, ctx)) != 0)
        return err;

/* We appear to have sucessfully received a record; increment the sequence number */
    if(rec->contentType != SSL_application_data_ssloppy)
          IncrementUInt64(&ctx->readCipher.sequenceNum);


#ifdef SSL_COMPRESSION
          if((ctx->compressNow) && (ctx->selectedCompression != NULL) &&
                    (ctx->selectedCompression->identifier != 0)) {

/* Allocate a buffer to return the plaintext in and return it */
                         if ((err = SSLAllocBuffer(&rec->contents, DEFAULT_BUFFER_SIZE,

                                 &ctx->sysCtx)) != SSLNoErr) {
                                         SSLFatalSessionAlert(alert_close_notify, ctx);
                                         return ERR(err);
                         }
                         if((err = ctx->selectedCompression->process(cipherFragment,


          &(rec->contents),


          ctx->readCompressRef,


          ctx)) != SSLNoErr) {
                                         SSLFreeBuffer(&rec->contents, &ctx->sysCtx);
                                         SSLFatalSessionAlert(alert_decompression_failure, ctx);
                                         return ERR(err);
                         }
#ifdef HYPER_DEBUG
                         fprintf(stderr, "Deompression created output of %d from size %d\n",
                                                        rec->contents.length,
  cipherFragment.length);
#endif
          } else {
                         if ((err = SSLAllocBuffer(&rec->contents, cipherFragment.length,

                                 &ctx->sysCtx)) != 0)
                         {
                                         SSLFatalSessionAlert(alert_close_notify, ctx);
                                         return ERR(err);
                         }
                         memcpy(rec->contents.data, cipherFragment.data, (size_t)
  cipherFragment.length);
          }
#else
          memcpy(rec->contents.data, cipherFragment.data, (size_t) cipherFragment.length);
#endif

    ctx->amountRead = 0;        /* We've used all the data in the cache */

    return SSLNoErr;
}

/* SSLWriteRecord does not send alerts on failure, out of the assumption/fear
 *  that this might result in a loop (since sending an alert causes SSLWriteRecord
 *  to be called).
 */
```

```
SSLErr
SSLWriteRecord(SSLRecord rec, SSLContext *ctx)
{   SSLErr      err;
    int         padding = 0, i, freerec = 0;
    WaitingRecord   *out, *queue;
    SSLBuffer       buf, payload, secret, mac, nonce;
    uint8           *progress;
    uint16      payloadSize,blockSize,nonceSize = 0;

    if (rec.protocolVersion == SSL_Version_2_0)
        return SSL2WriteRecord(rec, ctx);

    ASSERT(rec.protocolVersion == SSL_Version_3_0);
    ASSERT(rec.contents.length <= 16384);

#ifdef SSL_COMPRESSION
        if((ctx->compressNow) && (ctx->selectedCompression != NULL) &&
                (ctx->selectedCompression->identifier != 0)) {
                        SSLBuffer compdata;

                        /* make a guess about how long the buffer will need to be */
                        if((err = SSLAllocBuffer(&compdata, rec.contents.length + 4,

                         &ctx->sysCtx)) != SSLNoErr)
                                        return ERR(err);
                        if((err = ctx->selectedCompression->process(rec.contents, &compdata,


        ctx->writeCompressRef,


        ctx)) != SSLNoErr) {
                                        SSLFreeBuffer(&compdata, &ctx->sysCtx);
                                        return ERR(err);
                        }

                        rec.contents = compdata;
                        freerec = 1;
        }
#endif

    out = 0;
    /* Allocate a WaitingRecord to store our ready-to-send record in */
    if ((err = SSLAllocBuffer(&buf, sizeof(WaitingRecord), &ctx->sysCtx)) != 0)
        return ERR(err);
    out = (WaitingRecord*)buf.data;
    out->next = 0;
    out->sent = 0;

/* Allocate enough room for the transmitted record, which will be:
     *  5 bytes of header +
     *  encrypted contents +
     *  macLength +
     *  padding [block ciphers only] +
     *  padding length field (1 byte) [block ciphers only]
     */
    payloadSize = (uint16) (rec.contents.length + ctx->writeCipher.hash->digestSize);
    blockSize = ctx->writeCipher.symCipher->blockSize;
    if (blockSize > 0)
    {   padding = blockSize - (payloadSize % blockSize) - 1;
        payloadSize = (uint16)(payloadSize + padding + 1);
    }

  if(ctx->ssloppy)
  {
        /* in this case we need more room, for the nonce */
        nonceSize = (uint16) MAX(sizeof(uint64), ctx->writeCipher.symCipher->ivSize);
/*      payloadSize += nonceSize; decided this was wrong logic */
  }

    out->data.data = 0;
    if ((err = SSLAllocBuffer(&out->data, 5 + payloadSize + nonceSize,
                                                    &ctx->sysCtx)) != 0)
        goto fail;

    progress = out->data.data;
    *(progress++) = rec.contentType;
    progress = SSLEncodeInt(progress, rec.protocolVersion, 2);
    progress = SSLEncodeInt(progress, payloadSize, 2);

    /* Copy the contents into the output buffer */
    memcpy(progress, rec.contents.data, (size_t) rec.contents.length);
    payload.data = progress;
    payload.length = rec.contents.length;

    progress += rec.contents.length;
    /* MAC immediately follows data */
    mac.data = progress;
    mac.length = ctx->writeCipher.hash->digestSize;
    progress += mac.length;

  if(ctx->ssloppy)
  {
        uint64 noncevalue;

        if((err = SSLAllocBuffer(&nonce, nonceSize, &ctx->sysCtx)) != SSLNoErr)
                goto fail;
        if((err = ctx->sysCtx.random(nonce, ctx->sysCtx.randomRef)) != SSLNoErr)
                goto fail;

        memcpy(&noncevalue, nonce.data, sizeof(noncevalue));

        /* MAC the data, sloppy-style */
        if (mac.length > 0) /* Optimize away null case */
        {
                secret.data = ctx->writeCipher.macSecret;
                secret.length = ctx->writeCipher.hash->digestSize;
                if ((err = ComputeMAC(rec.contentType, payload, mac, noncevalue,
                                                    secret, &ctx->writeCipher, ctx)) != 0)
                        goto fail;
        }

        memcpy(progress, nonce.data, nonce.length);
        progress += nonce.length;

  }
  else
  {
        /* MAC the data, normal mode */
        if (mac.length > 0) /* Optimize away null case */
        {
                secret.data = ctx->writeCipher.macSecret;
                secret.length = ctx->writeCipher.hash->digestSize;
                if ((err = ComputeMAC(rec.contentType, payload, mac,
                                                    ctx->writeCipher.sequenceNum, secret,
                                                    &ctx->writeCipher, ctx)) != 0)
                        goto fail;
        }
  }

    /* Update payload to reflect encrypted data: contents, mac & padding */
    payload.length = payloadSize;

    /* Fill in the padding bytes & padding length field with the padding value; the
     *  protocol only requires the last byte,
     *  but filling them all in avoids leaking data
     */
    if (ctx->writeCipher.symCipher->blockSize > 0)
        for (i = 1; i <= padding + 1; ++i)
            payload.data[payload.length - i] = (uint8)padding;

    /* Encrypt the data */
    DUMP_BUFFER_NAME("cleartext data", payload);
   if ((err = ctx->writeCipher.symCipher->encrypt(payload, payload,
                                                    ctx->ssloppy ? &nonce:NULL,
                                                    ctx->writeCipher.symCipherState,
                                                    ctx))
   != 0)
          goto fail;

    DUMP_BUFFER_NAME("encrypted data", payload);

    /* Enqueue the record to be written from the idle loop */
    if (ctx->recordWriteQueue == 0)
        ctx->recordWriteQueue = out;
    else
    {   queue = ctx->recordWriteQueue;
        while (queue->next != 0)
            queue = queue->next;
        queue->next = out;
    }

   if(ctx->ssloppy)
          SSLFreeBuffer(&nonce, &ctx->sysCtx);
   else
          /* Increment the sequence number */
          IncrementUInt64(&ctx->writeCipher.sequenceNum);

   if(freerec)
          SSLFreeBuffer(&(rec.contents), &ctx->sysCtx);

    return SSLNoErr;

fail:   /* Only for if we fail between when the WaitingRecord is allocated and when it is
   queued */
    SSLFreeBuffer(&out->data, &ctx->sysCtx);
    buf.data = (uint8*)out;
    buf.length = sizeof(WaitingRecord);
    SSLFreeBuffer(&buf, &ctx->sysCtx);
          if(freerec)
                          SSLFreeBuffer(&(rec.contents), &ctx->sysCtx);
    return ERR(err);
}

static SSLErr
DecryptSSLRecord(uint8 type, SSLBuffer *payload, SSLContext *ctx)
{   SSLErr  err;
    SSLBuffer   content, nonce;

   if(type == SSL_application_data_ssloppy)
   {
          nonce.length = MAX(sizeof(uint64), ctx->readCipher.symCipher->ivSize);
          nonce.data = payload->data + (payload->length - nonce.length);
          payload->length -= nonce.length;
   }

    if ((ctx->readCipher.symCipher->blockSize > 0) &&
        ((payload->length % ctx->readCipher.symCipher->blockSize) != 0))
    {   SSLFatalSessionAlert(alert_unexpected_message, ctx);
        return ERR(SSLProtocolErr);
    }

    /* Decrypt in place */
    DUMP_BUFFER_NAME("encrypted data", (*payload));

   if(type == SSL_application_data_ssloppy)
   {
          if ((err = ctx->readCipher.symCipher->decrypt(*payload, *payload, &nonce, ctx-
   >readCipher.symCipherState, ctx)) != 0)
          {
```

```
                    SSLFatalSessionAlert(alert_close_notify, ctx);
                    return ERR(err);
            }
    }
    else
    {
            if ((err = ctx->readCipher.symCipher->decrypt(*payload, *payload, NULL, ctx-
    >readCipher.symCipherState, ctx)) != 0)
            {   SSLFatalSessionAlert(alert_close_notify, ctx);
        return ERR(err);
            }
    }
      DUMP_BUFFER_NAME("decrypted data", (*payload));

/* Locate content within decrypted payload */
      content.data = payload->data;
      content.length = payload->length - ctx->readCipher.hash->digestSize;
      if (ctx->readCipher.symCipher->blockSize > 0)
      {   /* padding can't be equal to or more than a block */
          if (payload->data[payload->length - 1] >= ctx->readCipher.symCipher->blockSize)
          {   SSLFatalSessionAlert(alert_unexpected_message, ctx);
              return ERR(SSLProtocolErr);
          }
          content.length -= 1 + payload->data[payload->length - 1];   /* Remove block size
    padding */
      }

/* Verify MAC on payload */
      if (ctx->readCipher.hash->digestSize > 0)       /* Optimize away MAC for null case */
          if(type == SSL_application_data_ssloppy)
          {
                  uint64 nonceNumber;

                  memcpy(&nonceNumber, nonce.data, sizeof(nonceNumber));
                  if ((err = VerifyMAC(type, content, payload->data + content.length,
                                                          nonceNumber, ctx)) != 0)
                  {
                          SSLFatalSessionAlert(alert_bad_record_mac, ctx);
                          return ERR(err);
                  }
          }
          else
          {
                  if ((err = VerifyMAC(type, content, payload->data + content.length,
                                                          ctx->readCipher.sequenceNum, ctx)) !=
    0)
                  {
                          SSLFatalSessionAlert(alert_bad_record_mac, ctx);
                          return ERR(err);
                  }
          }


    *payload = content; /* Modify payload buffer to indicate content length */

    return SSLNoErr;
}

static uint8*
SSLEncodeUInt64(uint8 *p, uint64 value)
{   p = SSLEncodeInt(p, value.high, 4);
    return SSLEncodeInt(p, value.low, 4);
}

static SSLErr
VerifyMAC(uint8 type, SSLBuffer data, uint8 *compareMAC, uint64 seqNo, SSLContext *ctx)
{   SSLErr      err;
    uint8           macData[MAX_DIGEST_SIZE];
    SSLBuffer       secret, mac;

    secret.data = ctx->readCipher.macSecret;
    secret.length = ctx->readCipher.hash->digestSize;
    mac.data = macData;
    mac.length = ctx->readCipher.hash->digestSize;

    if ((err = ComputeMAC(type, data, mac, seqNo, secret,
                    &ctx->readCipher, ctx)) != 0)
        return ERR(err);

    if ((memcmp(mac.data, compareMAC, (size_t) mac.length)) != 0)
        return ERR(SSLProtocolErr);

    return SSLNoErr;
}

static SSLErr
ComputeMAC(uint8 type, SSLBuffer data, SSLBuffer mac, uint64 seqNo, SSLBuffer secret,
            CipherContext *cipherCtx, SSLContext *ctx)
{   SSLErr      err;
    uint8           innerDigestData[MAX_DIGEST_SIZE];
    uint8           scratchData[11], *progress;
    SSLBuffer       digest, scratch;

#ifdef HYPER_DEBUG
  int i;
  fprintf(stderr, "Buffer: ");
  for(i = 0; i < data.length; i++)
          fprintf(stderr, "%02x ", data.data[i]);
  fprintf(stderr, "\n");

  fprintf(stderr, "sequenceno: ");
  for(i = 0; i < sizeof(uint64); i++)
          fprintf(stderr, "%02x ", (unsigned char) *((unsigned char *) &seqNo) + i);
  fprintf(stderr, "\n");

  fprintf(stderr, "Secret: ");
  for(i = 0; i < secret.length; i++)
          fprintf(stderr, "%02x ", secret.data[i]);
  fprintf(stderr, "\n");
#endif

    ASSERT(cipherCtx->hash->macPadSize <= MAX_MAC_PADDING);
    ASSERT(cipherCtx->hash->digestSize <= MAX_DIGEST_SIZE);
    ASSERT(SSLMACPad1[0] == 0x36 && SSLMACPad2[0] == 0x5C);

    if(cipherCtx->digestCtx.data == NULL) {
      if ((err = SSLAllocBuffer(&cipherCtx->digestCtx,
                        cipherCtx->hash->contextSize, &ctx->sysCtx))
     != 0)
         goto exit;
      cipherCtx->hash->create(cipherCtx->digestCtx);
    }

/* MAC = hash( MAC_write_secret + pad_2 + hash( MAC_write_secret + pad_1 + seq_num + type +
   length + content ) ) */
    if ((err = cipherCtx->hash->init(cipherCtx->digestCtx)) != 0)
        goto exit;
    if ((err = cipherCtx->hash->update(cipherCtx->digestCtx, secret)) != 0)    /* MAC secret */
        goto exit;
    scratch.data = SSLMACPad1;
    scratch.length = cipherCtx->hash->macPadSize;
    if ((err = cipherCtx->hash->update(cipherCtx->digestCtx, scratch)) != 0)   /* pad1 */
        goto exit;
    progress = scratchData;
    progress = SSLEncodeUInt64(progress, seqNo);
    *progress++ = type;
    progress = SSLEncodeInt(progress, data.length, 2);
    scratch.data = scratchData;
    scratch.length = 11;
    ASSERT(progress == scratchData+11);
    if ((err = cipherCtx->hash->update(cipherCtx->digestCtx, scratch)) != 0)   /* sequenceNo,
   type & length */
```

```
        goto exit;
    if ((err = cipherCtx->hash->update(cipherCtx->digestCtx, data)) != 0)  /* content */
        goto exit;
    digest.data = innerDigestData;
    digest.length = cipherCtx->hash->digestSize;
    if ((err = cipherCtx->hash->final(cipherCtx->digestCtx, digest)) != 0) /* figure inner digest */
        goto exit;

    if ((err = cipherCtx->hash->init(cipherCtx->digestCtx)) != 0)
        goto exit;
    if ((err = cipherCtx->hash->update(cipherCtx->digestCtx, secret)) != 0)    /* MAC secret */
        goto exit;
    scratch.data = SSLMACPad2;
    scratch.length = cipherCtx->hash->macPadSize;
    if ((err = cipherCtx->hash->update(cipherCtx->digestCtx, scratch)) != 0)    /* pad2 */
        goto exit;
    if ((err = cipherCtx->hash->update(cipherCtx->digestCtx, digest)) != 0)    /* inner digest */
        goto exit;
    if ((err = cipherCtx->hash->final(cipherCtx->digestCtx, mac)) != 0)    /* figure the mac */
        goto exit;

    err = SSLNoErr; /* redundant, I know */

exit:
    return ERR(err);
}
```